Three-Channel LLM Watermarking for Hangul Jamo Structure

TEAM릁: 강희지, 강재현, 박서현, 여운봉

Abstract

As large language models (LLMs) increasingly generate text that is comparable to human writing, the ability to identify the source of AI-generated content has become critically important. Moving beyond existing watermarking approaches that are English-centric, we suggest a novel framework that takes advantage of the structural characteristics of Hangul: the initial (Choseong, X), medial (Jungseong, Y) and final (Jongseong, Z) jamo as three independent information channels. Our adjusted watermarking method operates at the Logits Processor stage of LLM generation, decomposes candidate tokens into jamo in real-time, and applies independent watermarks on each channel. Our technical goal is to maximize both capacity and robustness while minimizing degradation in text quality. The proposed approach is expected to provide a practical tool for the transparent and safe use of LLM in Korean.

1 Introduction

1.1 Advances in LLMs and the Need for Watermarking

Generative models such as ChatGPT and LLAMA can now create text that is difficult to distinguish from human writing. Although these advances are impressive, they also carry significant risks, such as fake news generation and ambiguity in the ownership of academic output.

One fundamental mitigation is *text watermarking*: embedding subtle signals in generated text that are detectable algorithmically, but difficult to perceive by humans. When LLMs predict the next token, they select tokens based on probability scores; watermarking slightly modifies these probabilities to hide a pattern. This pattern can then be detected by algorithms to attest that the text was generated by an AI.

1.2 Limitations of Prior Work and the Need for Korean-Specific Approach

Most of the proposed LLM watermarking techniques are designed primarily in English. English as an isolating language contrasts with Korean, which is an agglutinative language where stems combine with particles and endings to realize grammatical functions. Tokenizers commonly used in English-centric pipelines, such as BPE (Byte-Pair Encoding) or WordPiece, often split Korean words inefficiently into subword units rather than meaningful morphemes; for example, the single token *"경복궁으로"* could be split into *"경복궁"* and *"으로"*.

This fragmentation increases the length of the sequence and complicates the application of watermarking algorithms that operate on tokens. Additionally, many widely used LLMs are trained predominantly in English, leading to poor performance in non-English languages such as Korean. The honorific language and various word-endings in Korean require language-specific approaches to fully capture contextual and cultural nuances.

To overcome these issues, we suggest tokenizer and model-architecture adjustments tailored to Korean, including morphological analysis combined with BPE or direct jamo-level approaches. Our ultimate aim is to develop a system that is contextually nuanced while establishing foundational methodologies for morphologically rich languages.

2 Proposed Study

We propose a novel three-channel watermarking framework that integrates existing watermarking ideas with the three-part structural decomposition of Hangul jamo. The key mechanism is to intercept candidate tokens at the LLM's next-token generation stage (the Logits Processor), decompose tokens into jamo in real-time (initial, medial, final), and insert independent watermarks into each channel.

2.1 3-Channel Jamo Scoring

(1) **Real-time jamo decomposition**: When the LLM outputs a logits array for the next-token prediction, select the top K candidate tokens by probability. For each candidate token, consider only the *final syllable* among its syllables; decompose that syllable into initial x, medial y, and final z jamo, and map each to unique integer values via a pre-defined lookup table S_{last}.

(2) **Three-channel separation and watermark application**: Treat the decomposed jamo triplet (x, y, z) as three parallel channels (Channel X, Y, Z). Using a secret key and a pseudo-random function (PRF), generate independent Green/Red lists per channel. For each candidate token, compute a score adjustment based on the information bits b intended for insertion. We design a hash function as follows:

$$\text{Hash}(x, y, z) = (x \cdot C_1 + y \cdot C_2 + z \cdot C_3) \bmod 2^k,$$

where k is the number of bits being inserted at once, and C_1, C_2, C_3 are distinct constants chosen (e.g., primes) to minimize collisions.

(3) **Final token selection**: Among the top K candidate tokens, select the token whose hash equals the target bit value b:

$$\text{Hash}(x_t, y_t, z_t) = b.$$

This procedure enables embedding up to k watermark bits per subword token; for example, $k = 2$ allows encoding 2 bits (0–3).

3 Methods and Experimental Design

3.1 Model Selection

To validate the proposed three-channel jamo watermarking method, we adopt a two-stage model testing:

- **Initial prototyping**: Use a lightweight model such as KoGPT2-small for rapid validation and debugging. This model integrates well with the Hugging Face ecosystem and allows testing the core LogitsProcessor logic with modest resources.
- **Final performance evaluation**: After method stabilization, evaluate effectiveness on high-performance models more akin to industrial deployments, e.g., Llama 3-8B or Mistral 7B fine-tuned on Korean data.
- **Training and evaluation data**: Utilize large-scale publicly available Korean corpora spanning multiple domains to measure general performance without domain bias.

3.2 Development Environment

To ensure reproducibility and efficient experimentation, the following environment is established:

- **Core frameworks**: Use Hugging Face `transformers` as the backbone for the generation pipeline and for customizing the LogitsProcessor. Additional libraries such as `datasets` and `accelerate` will be used for training, inference, and preprocessing.
- **Compute infrastructure**: Experiments are conducted in GPU-enabled environments to ensure stable execution and faster runtimes for large models.
- **Pre-/post-processing**: Implement a `JamoWatermarkProcessor` Python class that includes the real-time jamo decomposition and mapping modules, as well as ECC (error-correcting code) and LACS (Linguistics-Aware Channel Selection) logic. Integrate this class into the `transformers` generation pipeline.

HuggingFace Pipeline Pseudocode

```
from transformers import AutoModelForCausalLM, AutoTokenizer

# 1. Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained("kogpt2-small")
tokenizer = AutoTokenizer.from_pretrained("kogpt2-small")

# 2. ECC payload and instantiate JamoWatermarkProcessor
ecc_payload = "1011010..."  # (ECC applied)
jamo_processor = JamoWatermarkProcessor(
    ecc_payload=ecc_payload,
    mode='robustness'  # robustness mode
)

# 3. Generate watermarked text
outputs = model.generate(
    input_ids,
    max_length=256,
    logits_processor=[jamo_processor],  # core integration
    **generation_kwargs
)

# 4. Final detokenization
watermarked_text = tokenizer.decode(outputs[0])
```

3.3 Core Implementation: `JamoWatermarkProcessor`

We implement a `JamoWatermarkProcessor` by inheriting from Hugging Face's `LogitsProcessor` API, which hooks into the text generation process in real-time. The operating flow is as follows:

(1) **Real-time candidate jamo decomposition**: The LogitsProcessor is invoked immediately prior to token selection. Extract the top K candidate tokens (Top-K) from the logits distribution. For each candidate token string, decompose only the final syllable into initial x, medial y, and final z jamo.

(2) **Jamo-to-integer mapping**: Map each decomposed jamo to integer indices using a lookup table. For example, initial $x \in [0, 18]$, medial $y \in [0, 20]$, final $z \in [0, 27]$.

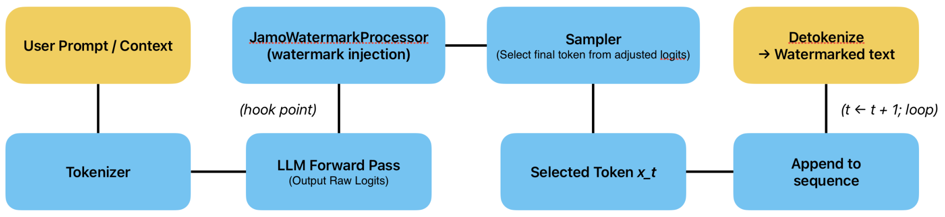
User Prompt / Context
JamoWatermarkProcessor (watermark injection)
Sampler (Select final token from adjusted logits)
Detokenize → Watermarked text
(hook point)
(t ← t + 1; loop)
Tokenizer
LLM Forward Pass (Output Raw Logits)
Selected Token x_t
Append to sequence

Figure 1: Simplified Diagram of Watermarking Process

(3) **Hash-based scoring**: Compute a hash value for each candidate token using the channel selection mode and the target bits to insert.

(4) **Final token selection**: Among the top K candidates, boost logits of those whose computed hash matches the target bit value. If multiple candidates satisfy the condition, choose the one with the highest original probability to minimize degradation of text quality.

Watermarking Procedure

Preparation

1. **Original message bits**: Prepare the base information bits to insert (e.g., `1011`).

2. **ECC**: Apply an ECC (e.g., Reed–Solomon) to the original bits to obtain a longer, more robust bit stream (e.g., `1011010`). ECC enables recovery of watermark bits even when some tokens are modified by attacks, compensating for vulnerabilities introduced by Korean subword segmentation. This supports our robustness target (post-attack detection rate $> 80\%$).

3. **Start watermarking**: The `JamoWatermarkProcessor` consumes the ECC-encoded payload (e.g., `1011010`) and sequentially inserts k-bit segments into generated tokens.

Token generation loop (time step $t \rightarrow t + 1$)

1. **LLM forward pass**: Given the generated sequence to date, the LLM outputs raw logits over the vocabulary for the next token.

2. **JamoWatermarkProcessor injection**:

   - *Candidate extraction & jamo decomposition*: Extract top K candidates from the logits and decompose each candidate's final syllable into (x, y, z) and map to integers.
   - *LACS & hash computation*: Pull the next target bit(s) (e.g., `1`) from the ECC stream and compute hash values according to the selected mode:
     – **Quality mode** (target: PPL change $< 3\%$): Exclude the initial x (which is sensitive to semantic changes) and compute hash from medial and final only: $\text{Hash}(y, z) = \ldots.$
     – **Robustness mode** (target detection $> 85\%$): Use all three channels: $\text{Hash}(x, y, z) = \ldots.$
   - *Logits adjustment*: Upweight logits of candidates whose hash matches the target bit(s).

3. **Sampler**: The sampler selects the next token based on the adjusted logits. The chosen token implicitly encodes the target watermark bit(s).

4. **Append to sequence**: Append the selected token x_t to the generated sequence.
5. **Repeat**: Increment time step and continue with the next bit from the ECC stream.
6. **Detokenize**: Once generation terminates (e.g., EOS token), decode token IDs into human-readable "watermarked text."

The `JamoWatermarkProcessor` supports two operational modes: (1) *Quality mode*, which omits the initial (Choseong) channel and uses only medial (Y) and final (Z) channels for watermarking to preserve semantic quality; and (2) *Robustness mode*, which utilizes all three channels (X, Y, Z) to maximize information capacity and resilience. This Linguistics-Aware Channel Selection (LACS) concept enables trade-offs between text quality and watermark robustness.

Implementation Pseudocode

```
from transformers import LogitsProcessor

class JamoWatermarkProcessor(LogitsProcessor):
    """
    Watermark injector using Hangul jamo three channels (initial, medial, final).
    Inherits Hugging Face’s LogitsProcessor for real-time intervention.
    """

    def __init__(self, ecc_payload: str, mode: str = ’robustness’, k_bits: int = 2,
                 top_k: int = 50):
        self.payload = ecc_payload  # ECC-encoded bit stream
        self.step_t = 0             # current time step index
        self.mode = mode            # ’robustness’ or ’quality’
        self.k_bits = k_bits        # number of bits inserted per step
        self.top_k = top_k          # number of candidate tokens to consider

    def _decompose_jamo(self, token_str: str) -> (int, int, int):
        """
        Decompose the last syllable of token_str into (initial, medial, final) indices.
        """
        # ... (omitted)
        return x, y, z

    def _calculate_hash(self, x: int, y: int, z: int) -> int:
        """
        Compute hash value according to selected LACS mode.
        """
        C1, C2, C3 = 19, 21, 28  # example constants

        if self.mode == ’quality’:
            # quality mode: exclude initial (x)
            hash_val = (y * C2 + z * C3) % (2**self.k_bits)
        else:
            # robustness mode: use all three channels
            hash_val = (x * C1 + y * C2 + z * C3) % (2**self.k_bits)

        return hash_val
    def __call__(self, input_ids: torch.LongTensor, scores: torch.FloatTensor)
                -> torch.FloatTensor:
        """
        Main function invoked per token generation (watermark injection).
        """
        # 1. Extract target bits for this step from payload
        target_bits = int(self.payload[self.step_t * self.k_bits :
                           (self.step_t + 1) * self.k_bits], 2)

        # 2. Select top-K candidate tokens
        top_k_indices = scores.topk(self.top_k).indices[0]

        # 3. For each candidate, compute jamo hash (LACS)
        for token_idx in top_k_indices:
            token_str = tokenizer.decode(token_idx)
            x, y, z = self._decompose_jamo(token_str)
            token_hash = self._calculate_hash(x, y, z)

            # 4. Adjust logits
            if token_hash == target_bits:
                # Boost candidates matching the target bit
                scores[0, token_idx] += 10.0  # additive boost example
            else:
                # Penalize mismatching candidates
                scores[0, token_idx] = -float('Inf')

        self.step_t += 1
        return scores
```

Unicode-based jamo extraction

All Hangul syllables (e.g., “가”, “각”, “힣”) are ordered sequentially in Unicode starting from “가” (U+AC00). The relative index of a syllable is:

$$\text{relative\_code} = \text{ord(syllable)} - \text{HANGUL\_START\_CODE}.$$

This index decomposes as:

$$\text{relative\_code} = x \cdot 588 + y \cdot 28 + z,$$

where $588 = 21 \times 28$. Thus:

$$x = \left\lfloor \frac{\text{relative\_code}}{588} \right\rfloor, \quad y = \left\lfloor \frac{\text{relative\_code mod } 588}{28} \right\rfloor, \quad z = (\text{relative\_code mod } 28).$$

```
// --- constants ---
HANGUL_START_CODE = 44032  // (0xAC00)
JUNGSEONG_X_JONGSEONG_COUNT = 588
JONGSEONG_COUNT = 28

FUNCTION get_jamo_indices(syllable):
    relative_code = unicode_value(syllable) - HANGUL_START_CODE
    x_index = relative_code // JUNGSEONG_X_JONGSEONG_COUNT
    remaining_code = relative_code % JUNGSEONG_X_JONGSEONG_COUNT
    y_index = remaining_code // JONGSEONG_COUNT
    z_index = remaining_code % JONGSEONG_COUNT
    RETURN (x_index, y_index, z_index)

// Example:
(x, y, z) = get_jamo_indices('크')
PRINT (x, y, z)  // result: (15, 18, 0)
```

3.4 Detection Algorithm

Given a text suspected of being watermarked, decompose the entire text into syllable units and separate sequences of initial, medial, and final jamo. Independently test, using a Z-test, how closely each channel's jamo distribution matches the pre-defined Green List. If all three channels exhibit statistically significant biases consistent with the watermark, the text is judged to contain the watermark.

3.5 Evaluation Metrics and Targets

- **Quality (PPL impact)**: Target a relative change in perplexity (PPL) of less than 5% compared to unwatermarked text to minimize naturalness degradation. Under "quality mode," an improved target of $< 3\%$ PPL change is adopted as suggested by a mentor.
- **Detection Performance**: Maintain a false positive rate (FPR) below 1%. We will additionally evaluate True Positive Rate (TPR) and F1 under different detection thresholds using the MarkLLM toolkit (FPR settings such as 0.1%, 1%, 10%).
- **Robustness**: Measure detection rates after various paraphrasing, summarization, and back-translation attacks; aim for $> 80\%$ post-attack detection. For *robustness mode*, an enhanced target of $> 85\%$ is set.
- **Embedding impact**: In response to practical deployment concerns (e.g., RAG services), measure the *embedding similarity change* before and after watermark insertion (e.g., using bge-m3 embeddings) via cosine similarity to quantify potential search-quality impact.

We will use the MarkLLM Toolkit for standardized benchmarking, which provides 12 canonical attack scenarios (paraphrasing, synonym substitution, Dipper, back-translation, etc.) to compare the proposed three-channel method against existing English-centric methods (e.g., KGW, Christ family).

4 Expected Outcomes and Significance

By incorporating Hangul's structural characteristics into watermarking, this research is expected to present a novel paradigm. Encoding up to three bits per token (one per jamo channel) can significantly increase capacity relative to prior approaches. The three independent channels also improve robustness: even if one channel's signal is degraded by attack, residual signals in other channels may enable detection. Practically, we intend to release the results as an open-source framework to mitigate misuse of LLMs and to increase transparency of AI-generated content.

We expect the method to be applicable across models built on varied tokenizers, and to be practical in real-world services (including RAG) due to design choices that optimize for subword boundary handling and LogitsProcessor efficiency.

5 Project Schedule

Activity period	Details
Weeks 1–4	Literature review of existing watermarking research and proposal refinement.
Weeks 5–8	Model and environment setup: select LLM model, configure GPU and Hugging Face-based development environment.
Weeks 9–11	Implement core modules and initial testing: implement the three-channel jamo LogitsProcessor and run preliminary PPL tests.
Week 12	Robustness experiments: run 12 standardized attacks from MarkLLM (Dipper, Synonym Substitution, etc.), perform Z-tests and ECC-based statistical analyses.
Weeks 13–14	Final analysis and prepare report and presentation materials including various charts.

6 Team Members and Roles

- **강희지**: Overall project leading and algorithm design. Establish the three-channel watermark model, hash-based token scoring logic, and implementation within the LogitsProcessor.
- **강재현**: Infrastructure and model optimization. Select final LLMs (Llama 3, Mistral, etc.), fine-tune on Korean data, build GPU/AWS infrastructure, and monitor/optimize resource usage.
- **박서현**: Data pre-/post-processing and module implementation. Implement real-time decomposition and integer mapping for token final syllables, and securely manage PRF-based Green/Red list keying and related debugging tools.
- **여운봉**: Experimental design and automation. Build pipelines to measure PPL, FPR, and robustness, design attack scenarios (paraphrasing, summarization, back-translation), and systematically record detection rates per scenario.

7 References

1. Dathathri et al. (2024). Scalable watermarking for identifying large language model outputs.
2. Gibney, E. (2024). Google unveils invisible 'watermark' for AI-generated text. *Nature*. `https://www.nature.com/articles/d41586-024-03462-7`
3. Google DeepMind. `synthid-text`. GitHub. `https://github.com/google-deepmind/synthid-text`
4. Google DeepMind. SynthID website. `https://deepmind.google/science/synthid/`
5. Google. Gemini API. `https://ai.google.dev/gemini-api/docs/tokens`
6. Google, SentencePiece. GitHub. `https://github.com/google/sentencepiece`
7. Hugging Face. LLM Course: Chapter 6. `https://huggingface.co/learn/llm-course/chapter6/`
8. Hugging Face. Introducing SynthID Text. Hugging Face Blog. `https://huggingface.co/blog/synthid-text`

9. Kirchenbauer et al. (2023). A Watermark for Large Language Models.

10. Park et al. (2020). An Empirical Study of Tokenization Strategies for Various Korean NLP Tasks.

11. Hugging Face. LLaMA. Transformers documentation. `https://huggingface.co/docs/transformers/v4.42.4/model_doc/llama`

12. OpenAI. tiktoken. GitHub. `https://github.com/openai/tiktoken`

13. Bryce et al. "An Introduction to Natural Language Processing Using Deep Learning". `https://wikidocs.net/book/2155`